VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Mean Shift segmentace obrazu pomocí technologie NVIDIA CUDA

Mean Shift Image Segmentation Method using NVIDIA CUDA Technology

2013 Tarek Batiha

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tarek Batiha**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Mean Shift segmentace obrazu pomocí technologie NVIDIA CUDA
Mean Shift Image Segmentation Method using NVIDIA CUDA Technology

Zásady pro vypracování:

Tato práce se bude zabývat segmentační metodou Mean Shift a jejím urychlením pomocí GPU (technologie NVIDIA CUDA). Výsledkem bude práce pojednávající o technologii NVIDIA CUDA, jejích možnostech a zásadách práce s ní, aplikované akceleraci na segmentační metodě Mean Shift a nebude chybět srovnání rychlosti i náročnosti programování obou implementací.

Úkoly:
1. Nastudovat segmentační algoritmus Mean Shift a naprogramovat jej v prostředí C++.
2. Nastudovat technologii NVIDIA CUDA a naprogramovat v ní segmentační algoritmus Mean Shift
3. Popsat segmentační algoritmus Mean Shift.
4. Popsat technologii NVIDIA CUDA a její vlastnosti včetně použitých konstrukcí nutných k úspěšnému naprogramování tohoto algoritmu v prostředí CUDA (tedy zmínění problémů, se kterými se student při programování setkal a jejich řešení/vysvětlení).
5. Provést měření rychlostí obou implementací a prezentovat je ve vhodně formě
6. Vyvodit závěry o užitečnosti NVIDIA CUDA, zmínit rozdíly v práci pro programátora

Seznam doporučené odborné literatury:

[1] NVIDIA CUDA Programming Guide
[2] http://drdobbs.com/cpp/207200659
[3] Comaniciu, D., Meer, P.: Mean shift: a robust approach toward feature space analysis.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Milan Šurkala**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě, dne 6. 5. 2013 .....

Touto cestou bych rád poděkoval vedoucímu mé práce Ing. Milanu Šurkalovi za pomoc a cenné připomínky při zpracování bakalářské práce.

Abstrakt

Člověk je při pohledu na obraz schopen přirozeně rozeznat jednotlivé jeho části a objekty na něm zobrazené. Počítač toho schopen není, proto je vhodné implementovat nějakou metodu, která to umožňuje. Segmentace obrazu je skupina metod vedoucí k rozdělení digitálního obrazu na segmenty postavených na různých principech. Jednou z těchto metod je Mean Shift segmentace obrazu, která se řadí mezi shlukovací metody a je výpočetně hodně náročná. Cílem této bakalářské práce je seznámení se s touto metodou segmentace a její implementací, jak v klasickém prostředí jazyka C/C++, tak s využitím masivního paralelismu technologie NVIDIA CUDA. Součástí práce je srovnání výkonu různých implementací a srovnání náročnosti jejich implementace.

Klíčová slova: mean shift, CUDA, segmentace obrazu

Abstract

Man looking at the picture is naturally able to recognize its individual parts and objects displayed. The computer is not capable of doing so, because of it its is appropriate to implement a method that makes it possible. Image segmentation is a group of methods leading to the distribution of digital image into segments built on different principles. One of these methods is the Mean Shift image segmentation, which is one of the clustering methods and its computationally very demanding. The aim of this thesis is to get familiar with this method of segmentation and its implementation, as in the classic C / C + + language, as using massive parallelism NVIDIA CUDA technology. Part of this work is to compare the performance of different implementations and compare their implementations performance.

Key words: mean shift, CUDA, image segmentation

Seznam zkratek

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

GPU – Graphic Processing Unit

PC – Personal Computer

OS - Operating System

RAM - Random-access emory

Obsah

1 Úvod ... 6
2 Segmentace obrazu ... 7
2.1 Úvod ... 7
2.2 Prahování ... 7
2.3 Shlukovací metody ... 7
2.4 Regionální metody ... 7
2.5 Znalostní metody ... 7
2.6 Detekce hran ... 7
3 Mean Shift segmentace obrazu ... 8
3.1 Mean Shift obecně ... 8
3.2 Definice ... 8
3.2.1 Odhad hustoty ... 9
3.2.2 Vektor posunu ... 9
3.3 Kernel ... 9
3.3.1 Uniformní kernel ... 9
3.3.2 Epanechnikov kernel ... 9
3.3.3 Gaussian kernel ... 10
3.4 Faktory ovlivnující výkon ... 10
3.4.1 Kernel ... 10
3.4.2 Jasová vzdálenost ... 10
3.4.3 Poloměr výpočetního okna ... 11
3.4.4 Konvergence ... 11
4 CUDA ... 12
4.1 Úvod ... 12
4.2 Popis NVIDIA CUDA ... 12
4.3 Architektura GPU ... 13
4.4 Programový model ... 14
4.4.1 Kernel ... 14
4.4.2 Hierarchie vláken ... 14
4.4.3 Warp ... 15
4.5 Paměťový model ... 15
4.5.1 Globální paměť ... 15
4.5.2 Sdílená paměť ... 15
4.5.3 Lokální paměť ... 15

4.5.4 Konstantní paměť 15
4.5.5 Paměť textur 16
4.5.6 Registry 16
4.6 Průběh CUDA programu 16
4.7 Rozšíření jazyka CUDA C 16
5 Implementace 18
5.1 Použité nástroje 18
5.2 Popis programu 18
5.3 Použité technologie 18
5.3.1 OpenCV 18
5.3.2 OpenMP 19
5.3.3 CUDA 19
5.4 CPU implementace 19
5.4.1 Popis 19
5.4.2 Algoritmus 19
5.5 CUDA implementace 20
5.5.1 Rozdíl implementace CPU x GPU 20
5.5.2 Kernel 1 20
5.5.3 Kernel 2 21
5.6 Nekonečný kernel 23
5.6.1 CPU implementace 23
5.6.2 CUDA implementace 1 23
5.6.3 CUDA implementace 2 24
6 Dosažené výsledky 25
6.1 Testovací podmínky 25
6.2 Výsledky segmentace 25
6.3 Vliv parametrů na výkon 26
6.4 Výkon konečného Epanechnikova kernelu 26
6.5 Výkon nekonečného Gaussianova kernelu 27
7 Závěr 29
8 Literatura 30

Seznam obrázků

Obrázek 1 Ukázka iterací 8
Obrázek 2 Uniformní kernel 9
Obrázek 3 3.3.2 Epanechnikov kernel 10
Obrázek 4 3.3.3 Gaussian kernel 10
Obrázek 5 Rozdíl CPU a GPU architektury 12
Obrázek 6 SMX GK110 13
Obrázek 7 4.4.2 Hierarchie vláken 14
Obrázek 8 Optimální načítání z paměti 21
Obrázek 9 Pararelní součet 22
Obrázek 10 Segmentace s různými parametry 25
Obrázek 11 Vliv parametrů na výkon 26
Obrázek 12 Výkon Epanechnikova kernelu 27
Obrázek 13 Výkon Nekonečného Gaussianova kernelu 28

Seznam tabulek

Tabulka 1 Testovací sestava 25

Seznam výpisu zdrojového kódu

Výpis 1 Ukázka funkce doMeanShift……..…...19
Výpis 2 Ukázka paralelizace CPU kódu……..…20
Výpis 3 Ukázka volání kernelu……...21
Výpis 4 Ukázka operací prvního vlákna bloku……..…….....21
Výpis 5 Ukázka součtu sum……....21
Výpis 6 Ukázka algoritmu…….……23
Výpis 7 Ukázka načítání z globální paměti…….……...23
Výpis 8 Ukázka načítání do sdílené paměti……..……..24

1 Úvod

V dnešní moderní době využíváme informační technologie na každém kroku. Každodenní běžné úkony většinou zahrnují ve větší či menší míře počítač. Dnešní počítače jsou schopné řešit složité rovnice nebo třeba předpovídat počasí. Pro člověka to tak jednoduché není. Na druhou stranu při pohledu na obraz je člověk schopen přirozeně rozlišit jednotlivé objekty. Pro počítač je obraz jako takový v podstatě jen spousta pixelů, které jsou definovány svými souřadnicemi a barvou. Segmentace obrazu je disciplína, která nám tyto body rozděluje na větší části, segmenty. Když takto rozdělíme obraz na větší celky, je pak jednodušší s nimi dále strojově pracovat a pokusit se například rozeznávat konkrétní objekty. Existuje mnoho druhů metod segmentace obrazu a fungují na různých principech. Jedním z algoritmů používaných k segmentaci obrazu je metoda Mean Shift. Tento algoritmus neočekává žádné vstupy a řadí se mezi takzvané clusterovací metody segmentace. Mean Shift je výpočetně velice náročný.

Není tomu tak dávno, kdy osobní počítače obsahovaly procesor s jedním jádrem. Zvyšování výkonů těchto procesorů se dosahovalo zdokonalováním architektury a zvyšováním frekvence. V nedaleké minulosti se začaly objevovat první procesory s více jádry. V dnešní době jsou naprosto běžné telefony s více jádrovými CPU. Na druhou stranu grafické karty byly vždy navrhovány k paralelnímu zpracování obrazu a obsahovaly proto spousty jednodušších výpočetních jader. Koncem roku 2006 přišla firma NVIDIA s technologií, která umožňovala využití masivního paralelismu grafických karet k běžným výpočtům pomocí programového rozšíření jazyka C/C++, technologií NVIDIA CUDA. Toto se ukázalo jako velice efektivní a v dnešní době nejvýkonnější superpočítače světa jsou vybaveny kartami s podporou této technologie.

Cílem této bakalářská práce je implementace Mean Shift algoritmu v prostředí NVIDIA CUDA. V této práci je popsána implementace tohoto algoritmu v prostředí C/C++ a v prostředí NVIDIA CUDA. Dále se v práci zabývám stručným popisem segmentace obrazu a pár vybraných metod segmentace.

2 Segmentace obrazu

2.1 Úvod

Segmentace obrazu představuje důležitý krok k analýze obsahu zpracovávaných obrazových dat. Cílem segmentace je rozdělit obraz na vzájemně se nepřekrývající oblasti, takzvané segmenty, což jsou části obrazu, které reprezentují konkrétní objekty v obraze. V podstatě jde o to, že každému pixelu v obraze je přiřazen index segmentu, ke kterému pixel náleží. Pixely se stejným indexem segmentu pak reprezentují nějaký objekt v obraze, což nám zjednodušuje celý obraz i jeho analýzu. Cílem segmentace obrazu je tedy zjednodušení reprezentace obrazu tak, aby bylo jednodušší jeho strojové zpracování. Praktické využití segmentace obrazu je především v počítačovém vidění, ve zpracování lékařských obrazových dat nebo v systémech řízení dopravy.

2.2 Prahování

Prahování je nejjednodušší metoda segmentace. U této metody je nejdůležitější zvolení správné hodnoty prahu, na jehož základě se pak vytváří z obrazu v odstínech šedi obraz binární. Prahování funguje v podstatě tak, že pro každý pixel se porovná jeho hodnota jasu s hodnotou prahu. Je-li jas větší nebo roven prahu, přiřadí se bodu jednička, v opačném případě nula. Tato metoda má lineární složitost.

2.3 Shlukovací metody

Vícerozměrná statistická metoda, také nazývaná jako *clusterová analýza*, při které se body shlukují tak, aby si v jednotlivých clusterech byly podobnější než body z různých clusterů. Každý pixel obrazu je reprezentován vektorem vlastností $v = [v1, v2,...,vn]$, což jsou např. jas, pozice atd. Hledají se takové vlastnosti, které mají body nacházející se v jedné oblasti stejné a v různých oblastech různé. Hledání shluků probíhá iterativně, takzvaná metoda učení bez učitele.

2.4 Regionální metody

Tyto metody jsou založeny na hledání společných vlastností pixelů jako například jas. Nejprve jsou v obraze rozmístěny pixely, a to buď náhodně, nebo iterativně. Jednotlivé segmenty pak vznikají rozrůstáním rozmístěných pixelů. Nevýhoda je, že při různém rozmístění pixelů, lze očekávat různé výsledky.

2.5 Znalostní metody

Mezi tyto metody patří všechny algoritmy, které využívají k segmentaci dříve získané znalosti o objektech. Vyhledávají se tedy známé objekty srovnáváním se vzorem. Vyhledávaný objekt se většinou nemůže absolutně shodovat se vzorem, kvůli různým šumům, zkreslení atd. Hledá se proto maximum shody vhodného kritéria a to i pro obrazy natočené, či v různém měřítku.

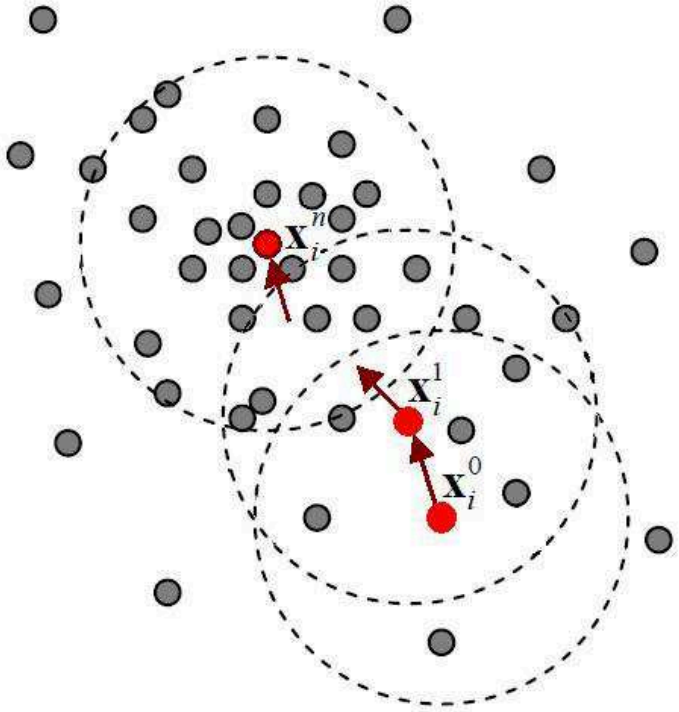
2.6 Detekce hran

Tato metoda hledá oblasti pixelů, kde se výrazně mění jas, takzvané hrany. Hranu můžeme nalézt na hranicích objektů, na rozhraní světla a stínu – takzvaná *skoková* hrana**,** nebo v místech trojrozměrných hran objektů – takzvaná *trojúhelníková* hrana. Hrany nemusí vznikat jen mezi objekty ve scéně. Hrany detekujeme tak, že v místě hrany očekáváme velkou derivaci jasové funkce. Nejvyšší hodnota derivace bude směřovat kolmo na hranu. Kvůli zjednodušení výpočtu se hrany detekují ve dvou nebo čtyřech směrech. Vzhledem k vlastnostem této metody mohou vznikat falešné nebo přerušované hranice. Výsledný obraz je proto nutné dále zpracovat.

3 Mean Shift segmentace obrazu

3.1 Mean Shift obecně

Mean Shift segmentace obrazu je algoritmus, který patří mezi takzvané shlukovací metody segmentace. Řadí se mezi bezparametrické techniky analýzy prostoru, proto nepotřebuje znát předem počet nebo tvar segmentů. Patří mezi clusterovací metody segmentace, pracuje iterativně. Význam slov Mean shift lze volně přeložit jako přesun za průměrem. V podstatě se jedná o to, že se v každé jednotlivé iteraci vypočítá vážený průměr jednotlivých bodů výpočetního okna, a do tohoto průměru se okno v následující iteraci přesune. Takto algoritmus rekurzivně pokračuje, dokud nedosáhne konvergence. Konvergencí je myšlen nulový posun v další iteraci. Jelikož posun může být velice malý, prakticky v rámci jednoho bodu obrazu, je vhodné ukončit ho před dosažením konvergence. Například definováním maximálního možného počtu iterací nebo minimální délkou posunu. Všechny tyto operace probíhají nad stejnými zdrojovými daty. Existuje i modifikace Mean Shift algoritmu, takzvaný Bluring Mean Shift, který se liší od klasického Mean Shiftu tím, že nepracuje nad stejnými zdrojovými daty. V každé další iteraci pracuje nad výslednou množinou předcházejících iterací.



Obrázek 1 Ukázka iterací

3.2 Definice

Mean shift segmentace shlukuje body na základě jejich jasu a blízkosti jejich souřadnic. Jednotlivé body obrazu jsou reprezentovány vektorem $x_i = [x, y, z]$, kde x a y jsou 2D souřadnice bodu v obraze a z je hodnota jasu. Je možné pracovat i s více rozměrným prostorem např. 5D, kde poslední tři souřadnice budou reprezentovat jednotlivé barevné složky. Algoritmus má tři parametry. První parametr je velikost poloměru výpočetního okna h, druhým radiálně symetrická funkce tzv. kernel K, třetím pak maximální jasová vzdálenost.

3.2.1 Odhad hustoty

Prvním krokem Mean Sift segmentace je odhad hustoty bodů, který je dán následujícím vztahem:

$$p(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

pro $h > 0$, kde K reprezentuje radiálně symetrickou funcki, kernel.

3.2.2 Vektor posunu

Vektor posunu je pak dán vztahem:

$$m(x) = \frac{\sum_{i=1}^{n} x_i K\left(\left\|\frac{x - x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} K\left(\left\|\frac{x - x_i}{h}\right\|^2\right)} - x$$

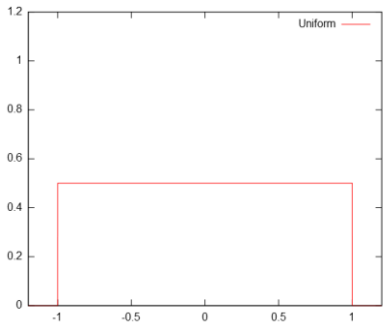
Jedná se o rozdíl mezi váženými posuny, kde kernel K určuje váhu bodů a x je střed kernelu.

3.3 Kernel

Kernel je radiálně symetrická funkce, která určuje váhu, jakou se budou započítávat jednotlivé body do výpočetního okna. Rozhoduje se na základě jejich prostorové vzdálenosti od středu výpočetního okna a jasové vzdálenosti bodu od středu. Kernel tedy přímo ovlivňuje výsledek i rychlost algoritmu. Existuje mnoho druhů kernelů, uvedu zde tři nejznámější.

3.3.1 Uniformní kernel

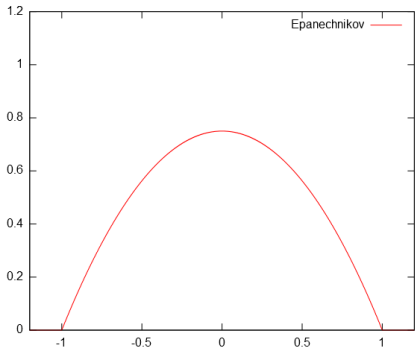
Nejjednodušší, nejméně výpočetně náročný. Je dán vztahem $K(u) = \frac{1}{2} * 1\{|u| \leq 1\}$.



Obrázek 2 Uniformní kernel

3.3.2 Epanechnikov kernel

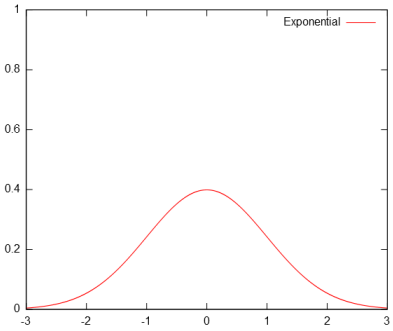
Výpočetně náročnější, ale poskytuje kvalitnější výsledky než uniformní kernel. Je dán vztahem $K(u) = \frac{3}{4} * (1 - u^2)\{|u| \leq 1\}$.



Obrázek 3 3.3.2 Epanechnikov kernel

3.3.3 Gaussian kernel

Je dán vztahem $K(u) = \frac{1}{\sqrt{2*\pi}} * e^{-\frac{1}{2}*u^2}$. Můžeme si všimnout, že definiční obor nemá omezení.



Obrázek 4 3.3.3 Gaussian kernel

3.4 Faktory ovlivňující výkon

Jak už jsem zmínil v kapitole 3.2, Mean Shift segmentace má tři paramenty. Tyto parametry poměrně hodně ovlivňují výkon algoritmu a celkový výsledek segmentace.

3.4.1 Kernel

Kernel má nejmenší vliv na výkon. Algoritmus vykonává hodně jednoduchých operací, a proto například při volbě Epanechnikova kernelu se musí ošetřit podmínkou definiční obor a obsahuje exponent, což nám určitě ovlivní rychlost algoritmu.

3.4.2 Jasová vzdálenost

Dalším parametrem ovlivňujícím výkon je maximální hodnota jasové vzdálenosti mezi bodem ve středu výpočetního okna a dalším libovolným bodem. Pracujeme-li s obrazem jako s 3D prostorem, kde máme jeden parametr vzhledu bodu, a tím je hodnota jasu, stačí nám vypočítat pouze rozdíl těchto jasů. Pracujeme-li ale s obrazem jako s 5D prostorem, kde je vzhled bodu definován třemi barvami, je nutné zjistit podobnost těchto bodů, například výpočtem euklidovské vzdálenosti mezi těmito body.

Jasová vzdálenost má velký vliv na výsledek segmentace. Přímo ovlivňuje výsledný tvar segmentů a tím i celkovou rychlost výpočtu, protože je tím přímo ovlivněn počet iterací.

3.4.3 Poloměr výpočetního okna

Největší dopad na výkon má velikost výpočetního okna. Tento parametr určuje maximální vzdálenost, mezi aktuálně zpracovávaným bodem a jeho okolím. Body okolí, které spadají do této vzdálenosti, se musí zpracovat, tj. načíst z paměti a vyhodnotit. Tento parametr má největší vliv na výkon Mean Shift segmentace. Má také velký vliv na výsledek segmentace.

3.4.4 Konvergence

Vzhledem k tomu, že algoritmus je iterativní a končí, až dosáhne konvergence, mohou počty iterací nabývat astronomických rozměrů. Konvergence dosáhne algoritmus v momentě, kdy je vektor posunu roven nule. Vzhledem k tomu, že je nutné v každé iteraci vypočítat velikost posunu, je vhodné ukončit algoritmus při velice malých posunech. Kupříkladu, je-li posun menší, než je tisícina pixelu, můžeme iterace ukončit bez většího vlivu na výsledek. Je také vhodné určit maximální počet iterací, kterého může jeden bod dosáhnout. Překročí-li se tato hodnota, uloží se hodnota vypočtená v poslední iteraci.

4 CUDA

4.1 Úvod

Dnešní moderní grafické karty ve srovnání s moderními procesory dosahují až několikanásobně vyššího výkonu. Tento výkonnostní rozdíl je dán rozdílnou architekturou těchto zařízení. Grafické karty obsahují až stovky výpočetních jader, zatímco počty v procesorech jsou v řádech jednotek až desítek. Tento rozdíl je dán různým určením těchto zařízení. GPU jsou navržená na provádění stovek operací souběžně, které jsou důležité v oblasti zpracování 3D grafiky nebo počítačových her. CPU jsou na druhou stranu navržená k sériovému zpracování, obsahují výkonná výpočetní jádra a nízké latence. I když moderní CPU jsou navržená k běhu více vláken, stále důležitý je však výkon jednoho jádra. Jednak proto, že některé algoritmy nelze implementovat paralelně a také proto, aby se zachovala kompatibilita se staršími programy, které byly navrženy pro běh v jednom vlákně. V dnešní době je výkon GPU několikanásobně větší než CPU, a proto se začaly grafické karty využívat i pro jiné než grafické výpočty. Využívání GPU pro jiné než grafické výpočty se označuje zkratkou GPGPU, General-purpose computing on graphics processing units, volně přeloženo jako „víceúčelové počítání na grafických procesorech“. NVIDIA CUDA je nejrozšířenější proprietární rozhraní pro výpočty na GPU.

4.2 Popis NVIDIA CUDA

V listopadu roku 2006 společnost NVIDIA uvedla platformu CUDA (Compute Unified Device Architecture), univerzální výpočetní platformu a programovací model. Tato platforma je dostupná pouze pro GPU společnosti NVIDIA, jedná se tedy o proprietární technologii. Konkurenční technologie firmy AMD se nazývá AMD APP. Pro obě tyto platformy je možné psát programy ve frameworku OpenCL.

Platforma CUDA umožňuje řešit komplexní výpočetní problémy mnohem efektivnějším způsobem, než na CPU. Také je schopna spouštět na GPU programy napsané v jazycích C/C++, FORTRAN a jiných, dále programy postavené na technologiích jako DirectCompute nebo OpenACC.

Základní vlastnosti technologie CUDA:

- Výrazně zjednodušuje programování v GPGPU
- Je založena na rozšíření jazyka C/C++
- Vysoký výkon paralelních aplikací
- Funguje pouze na GPU NVIDIA

Obrázek 5 Rozdíl CPU a GPU architektury

4.3 Architektura GPU

Většinu plochy GPU zabírá velké množství jednoduchých skalárních procesorů. Tyto nejmenší jednotky jsou organizovány do větších celků, takzvaných streaming multiprocesorů. Obecně se multiprocesor skládá ze stream procesorů, pole registrů, load/store jednotek, SFU – special function unit – jednotka pro výpočet složitějších funkcí např. sin, cos, exp, a double precision unit – jednotka s dvojitou přesností. Dále se na streaming multiprocesor nachází sdílená paměť a warp plánovače.

Nejnovější produkt firmy NVIDIA je jádro GK110 architektury Kepler. Jedná se o čip vyvinutý speciálně pro HPC segment. Plné jádro Kepler GK110 se skládá z 15 streaming multiprocesorů a šesti 64 bitových paměťových řadičů. Jeden multiprocesor se skládá ze 192 single precision CUDA jader, 64 double-precision jednotek, 32 SFU a 32 load/store jednotek. Například výpočetní karta Tesla K20 postavená na jádře GK100 má teoretický výkon v single precision výpočtech 3,520 TFLOPs, v Double precision 1,170 TFLOPs a paměťovou propustnost 208 GB/s. Tato karta je mimo jiné jedním ze stavebních kamenů superpočítače Titan, který je momentálně nejvýkonnější na světě. Toto jen dokazuje efektivitu výpočtů na GPU.

Obrázek 6 SMX GK110

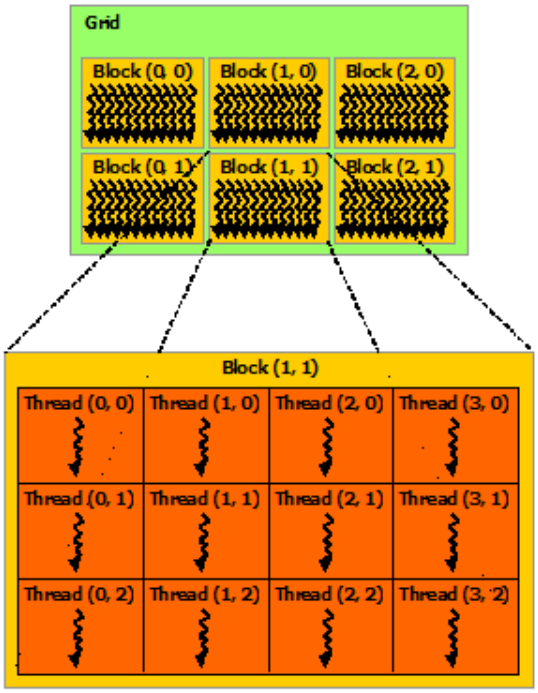
4.4 Programový model

4.4.1 Kernel

CUDA C rozšiřuje jazyk C o funkce nazývané kernely. Kernel obsahuje tu část kódu, která se bude provádět na GPU. Pro volání se zavádí nová syntaxe <<<*velikost mřížky, počet vláken*>>>(argumenty, …). Velikostí mřížky podrobněji vysvětlím v kapitole 4.4.2. Počet vláken udává, v kolika vláknech se má kód vykonat. Výpočty na GPU jsou specifické tím, že se ve všech vláknech provádí stejný kód, ale nad různými daty. Z toho nám vyplývá, že je vhodné použití GPU k výpočtu nad obrovskými poli, nebo maticemi prvků, kde se na každý prvek aplikuje stejná operace. Volání kernelu nemá žádnou návratovou hodnotu, proto je nutné si výsledná data ukládat do předem alokované paměti na kartě a pak si je zkopírovat do paměti RAM počítače viz kapitola 4.6.

4.4.2 Hierarchie vláken

Vzhledem k tomu, že se v každém vlákně provádí stejný kód, potřebujeme nějakou metodu identifikace jednotlivých vláken tak, abychom mohli například přistupovat v každém vlákně k jednotlivému prvku pole. V kernelu je k tomuto účelu vestavěná proměnná threadIdx. Je to třísložkový vektor, který jednoznačně identifikuje každé vlákno pomocí třírozměrného indexu. Vlákna je možné organizovat do jedno až tří rozměrných bloků. V tomto případě proměnná threadIdx identifikuje vlákno jen v rámci bloku, pro jednoznačnou identifikaci vlákna je třeba dopočítat pozici na základě rozměrů bloku a jeho indexu. Vlákna jsou nezávislá na ostatních a nemají možnost globální synchronizace. Toto ale neplatí v rámci jednoho bloku. Synchronizace vláken je možná v rámci jednoho bloku pomocí funkce __syncThreads(). Všechna vlákna v rámci jednoho bloku mají přístup do společné sdílené paměti, což je velice mocný nástroj pro zvýšení paměťové propustnosti vyvíjeného programu. Velikost bloku se určuje při volání kernelu. Pro tento účel je určená struktura dim3, pomocí které určíme rozměr bloku.



Obrázek 7 4.4.2 Hierarchie vláken

4.4.3 Warp

Vlákna v rámci jednoho bloku se neprovádějí všechna najednou, ale jsou rozdělená do menších jednotek, takzvané warpy. Velikost jednoho warpu je často 32, není to však podmínkou. Každý warp se provádí jako SIMD – všechna vlákna v rámci jednoho warpu musí vykonávat stejnou instrukci. Z toho vyplývá problém. Při větvení kódu podmínkami if else nastává degradace výkonu, protože vlákna v jednom warpu budou muset provádět různé instrukce. To znamená, že zatímco některá vlákna pracují, ostatní jsou v nečinnosti. Dojde ke zvýšení počtu instrukcí provedených warpem, a proto ke snížení výkonu programu.

4.5 Paměťový model

4.5.1 Globální paměť

Globální paměť představuje největší paměť na grafické kartě, velikost se pohybuje v řádu gigabajtů, nachází se na zařízení. Zařízení k této paměti přistupuje pouze v 32, 64 nebo 128 bajtových paměťových transakcích. Mějme modelovou situaci, kde chce jedno vlákno přistoupit k hodnotě v globální paměti, například 32 bitový datový typ integer z pole int a[128]. Vláknem hledaná hodnota se nachází na pozici a[7], zařízení proto provede 32 bajtovou transakci a načte prvních 8 hodnot z paměti. Toto je ale neefektivní, protože se načte dalších 7 hodnot zbytečně. Z tohoto důvodu je vhodné, aby například vlákna v rámci bloku četla z globální paměti paralelně a neprovádělo se proto zbytečné čtení. Další nevýhodou této paměti je její nízká datová propustnost oproti dalším typům paměti a fakt, že pro přístup do této paměti musí vlákna čekat několik strojových cyklů. Výhodou je naopak její velikost, a proto je vhodná k ukládání velkých datových struktur, jako například matice. Je přístupná všem vláknům.

4.5.2 Sdílená paměť

Sdílená paměť se nachází přímo na čipu. Proto je mnohem rychlejší a má mnohem menší latenci než globální nebo lokální paměť. Je rozdělena na části, takzvané banky, z nichž všechny jsou stejně velké. Jakékoliv požadavky na čtení nebo zápis může probíhat u všech bank současně. Nevýhodou ovšem je, když se pokouší přistupovat více vláken k jedné bance zároveň. Pak nastává takzvaný konflikt bank a požadavek se musí zpracovat serializovaně. Tato paměť se dá alokovat dvěma způsoby, a to buď staticky, nebo dynamicky. Při dynamické alokaci se udává požadovaná velikost sdílené paměti v bajtech jako třetí argument při volání kernelu, za velikost mřížky a počet vláken. Pří statické alokaci musí být velikost sdílené paměti známa už při překladu programu. Ideální způsob využití této paměti je, když vlákna paralelně načtou data z globální paměti do sdílené a poté s nimi pracují. Vhodným využitím sdílené paměti lze dosáhnout vysoké efektivity programu.

4.5.3 Lokální paměť

Lokální paměť je část globální paměti, kterou má pro sebe vyhrazené každé vlákno. Mohou se v ní nacházet některé lokální proměnné jako například dynamická pole, velké struktury, které se nevejdou do registrů, nebo když vlákno vypotřebuje všechny registry. Má vysoké latence a malou propustnost. Liší se od globálních pamětí tím, že je kešovaná v L1.

4.5.4 Konstantní paměť

Tato paměť se nachází v paměti zařízení. Je kešována, má pouze read only přístup. Můžou k ní přistupovat všechna vlákna. Při přístupu všech vláken najednou muže paměť vysílat broadcastem.

4.5.5 Paměť textur

Tato paměť se nachází v paměti zařízení. Je pouze read only. Je navržená pro přístup k 2D datům, takže nejlepšího výkonu lze dosáhnout, když budou vlákna v rámci jednoho warpu přistupovat k bodům blízko sebe. Je kešovaná, takže jen v případě že se hledaný bod nenachází v keši, přistupuje do globální paměti.

4.5.6 Registry

Registry se nacházejí přímo v čipu. Z toho vyplývají minimální latence a maximální datová propustnost. Jedná se v podstatě o nejrychlejší druh paměti. Nevýhoda registrů je jejich omezený počet na jedno jádro. Do registrů jsou nejčastěji ukládány lokální proměnné. Hodnoty uložené v registrech jsou přístupné pouze pro vlákno, které je vytvořilo. Hodnota v registru má životnost vlákna. Při překročení limitu registrů se ukládají proměnné do lokální paměti.

4.6 Průběh CUDA programu

Na rozdíl od programů, které ke svým výpočtům používají pouze klasické CPU a paměť RAM, mají programy naimplementované na platformě CUDA trochu rozdílný průběh. U klasického programu napsaného v jazyce C++ je možné provést výpočet v paměti RAM a výsledek zobrazit uživateli. Programy v CUDA mají následující průběh:

- Naalokuj paměť pro zpracovávané hodnoty a výsledek na zařízení (grafické kartě)
- Nakopíruj data z paměti RAM do naalokovaných částí paměti
- Spusť kernel
- Počkej na výsledek výpočtu
- Nakopíruj výsledek výpočtu do paměti RAM
- Uvolni alokovanou paměť

Z tohoto je patrné, že samotné spuštění výpočtu a získání výsledku si vyžádá určitou režii CPU a paměťové propustnosti. Proto není vhodné provádět výpočty nad malými daty na GPU, ale nechat tyto jednoduché úlohy na CPU. Na druhou stranu je velice vhodné využít nad velkými daty masivního paralelismu a výrazně tak zvýšit efektivitu programu nebo algoritmu.

4.7 Rozšíření jazyka CUDA C

Technologie CUDA doplňuje jazyk C o několik identifikátorů, kterými jsou:

Identifikátory funkcí:

- **__Device__** – tento identifikátor deklaruje funkci jako spustitelnou pouze na zařízení a lze ji volat jen z CUDA kódu.
- **__Global__** – tento identifikátor deklaruje funkci jako kernel. Tato funkce je spustitelná na zařízení, lze ji spouštět z hostitelského stroje. Tato funkce má vždy návratový typ void. Volání kernelu je asynchronní – to znamená, že funkce hned po zavolání vrátí návratový kód CPU, zatímco výpočet na GPU stále probíhá. Pomocí funkce cudaDeviceSynchronize() můžeme počkat na dokončení vykonáni kernelu.
- **__Host__** – tento identifikátor deklaruje funkci jako volatelnou a spustitelnou pouze na hostitelském zařízení.

Identifikátory proměnných:

- **__Device__** – tento identifikátor deklaruje proměnnou na zařízení. Proměnné tohoto typu jsou uloženy v globální paměti, přístupné všem vláknům a mají životnost aplikace.
- **__Constant__** – tento identifikátor deklaruje proměnnou na zařízení. Proměnná je konstanta, to znamená, že je uložená v konstantní paměti, má životnost aplikace a je přístupná všem vláknům.
- **__Shared__** – tento identifikátor deklaruje proměnnou, která má životnost bloku, ve kterém byla vytvořena. Proměnná se nachází ve sdílené paměti bloku a je přístupná pouze vláknům v rámci daného bloku. Pokud deklarujeme pole tohoto typu, máme dvě možnosti. První možnost je taková, že velikost pole je dána už při překladu. Druhá možnost je, že pole budeme alokovat dynamicky. V tomto případě je potřeba při volání kernelu zadat jako třetí parametr maximální velikost sdílené paměti v bitech. Tuto velikost pak nesmíme překročit. Při vytváření takového pole se použije klíčové slovo extern.

5 Implementace

V této části práce budu popisovat výsledky implementace Mean Shift algoritmu na různých platformách a to na x86 CPU a NVIDIA CUDA. Všechny zdrojové kódy popsané v dalších kapitolách se nachází na přiloženém CD.

5.1 Použité nástroje

- Nvidia Nsight Eclipse Edition – Upravená verze vývojového prostředí eclipse, rozšířená o překladač nvcc a podporu platformy CUDA C.

- Nvidia Visual Profiler – Tento nástroj je vhodný k ladění výkonu aplikace napsané v jazyce CUDA C. Nástroj analyzuje vliv programu na jednotlivé části gpu. Ukazuje například, kolik registrů využívají jednotlivá vlákna, celkové vytížení gpu. Nvidia Visual Profiler také analyzuje využití jednotlivých druhů paměti a jejich propustnost. Tento ukazatel je důležitým ukazatelem optimálního využití sdílné paměti a potažmo celkové efektivity aplikace.

- ImageMagick – Pro srovnání výsledků jednotlivých algoritmů jsem použil utilitu compare z balíku nástrojů ImageMagick. Tato utilita srovnává matematicky a vizuálně rozdíl mezi dvěma obrazy.

5.2 Popis programu

Program jsem implementoval jako konzolovou aplikaci. Toto rozhraní jsem zvolil proto, aby byly co nejefektivněji využity systémové prostředky a paměť stroje, na kterém program poběží. Program očekává při spuštění tři argumenty. Těmi jsou: obrázek, který chce uživatel zpracovat, velikost výpočetního okna a maximální rozdíl jasu.

Mnou navržený algoritmus načítá jak barevné obrázky, tak obrázky v odstínech šedi. Výsledný obrázek je po dokončení segmentace zobrazen uživateli a uložen.

Načtený obraz se zpracovává v odstínech šedi a tedy i výsledek obsahuje pouze hodnotu jasu. Tento způsob zpracování jsem zvolil proto, že na výsledný tvar segmentů to nemá velký vliv a výkonově je rychlejší, protože se pracuje pouze 3D prostorem. Většina obrazových dat používaných ve výpočetní technice je v barevném prostoru RGB, protože je to nejvhodnější k strojovému zobrazení. Tento barevný prostor však není vhodný k segmentaci obrazu, protože nemá mnoho společného s tím, jak člověka vnímá barvy. V praxi se využívá například barevný prostor YUV, který mnohem lépe koresponduje s reálným světem.

5.3 Použité technologie

5.3.1 OpenCV

K načítání zdrojových obrázků jsem využil knihovnu OpenCV. Tato knihovna je vydána pod BSD licencí a je tedy vhodná k akademickému i komerčnímu použití. Běží pod operačními systémy Windows, Mac, Linux a Android.

5.3.2 OpenMP

Pro paralelizaci CPU kódu jsem použil knihovnu OpenMP. Jedná se o API, které podporuje multiplatformní programování se sdílenou pamětí v jazycích C/C++ a FORTRAN na všech architekturách. OpenMP se spouští pomocí takzvaných direktiv.

5.3.3 CUDA

Viz kapitola 4.

5.4 CPU implementace

5.4.1 Popis

Pro načítání zdrojového obrazu jsem použil třídu Mat knihovny OpenCV. Obraz lze načítat buď v odstínech šedi, nebo barevně. Zvolil jsem odstíny šedi, viz 5.2. Vzhledem k tomu, že Mean Shift algoritmus pracuje nad stejnými zdrojovými daty, rozhodl jsem se vytvořit dvě instance třídy Mat, kde první reprezentuje zdrojová data, do druhé se pak ukládá výsledný obraz. Jedním z atributů třídy Mat je ukazatel na jednorozměrné pole, které obsahuje hodnoty jasů jednotlivých bodů. Pro co nejrychlejší zpracování pracuji v programu s tímto polem. V tomto případě, kdy jsem načetl zdrojový obraz v odstínech šedi, je pole typ unsigned char.

5.4.2 Algoritmus

Algoritmus funguje tak, že ve dvou cyklech se v poli prochází prvek po prvku a pro každý prvek se zavolá funkce doMeanShift, která provádí výpočet nové pozice. Funkce má tři argumenty. Těmi jsou x a y souřadnice bodu, které reprezentují proměnné cyklu a hodnota jasu bodu na těchto souřadnicích. Zdrojové a cílové pole jsou globální proměnné, není je proto potřeba předávat jako argument funkce. Funkce obsahuje tři vnořené cykly. Dva vnořené cykly provádějí výpočet sum okolí zpracovávaného bodu, nezanořený cyklus pak reprezentuje jednotlivé iterace. V nezanořeném cyklu se vypočítá posun bodu a v případě, je-li posun dostatečně velký, probíhá další iterace, kdy se jako nové souřadnice použijí vypočtené hodnoty vnořenými cykly. Je-li ale posun menší, než předem specifikovaná hodnota, uloží se vypočtená hodnota jasu do výsledného pole na souřadnicích předaných argumentem funkci a cyklus se ukončí.

```
1.  void doMeanShift( int xP, int yP, uint8_t color ){
2.    //proměnné
3.    while(true){
4.      //podmínky okraje obrazu a výpočetního okna
5.      for( int i = iA ; i < iB ; i++ )
6.      {
7.        for( int j = jA ; j < jB ; j ++)
8.        {
9.          if( (i - x) * (i - x) + (j - y ) * (j - y) <= radiusE2 )
10.         {
11.           //výpočet sum
12.         }
13.
14.       }
15.     }
16.     //ukončení algoritmu?
17.     if( iterace > maxIteraci || posun < minPosun ){
18.       //ulož výsledek
19.       break;
20.     }
21.     //nastaveni nových hodnot středu výpočetního okna
22.   }
23.
24. }
```

Výpis 1 Ukázka funkce doMeanShift

Pro paralelizaci tohoto algoritmu jsem použil technologii OpenMP. Vzhledem k tomu, že každé vlákno pracuje nezávisle na ostatních a ze sdíleného pole se reprezentující obraz pouze načítá, není třeba žádné proměnné zamykat.

```
1.  #pragma omp parallel for
2.  for( int i = 0 ; i < xImg ; i ++)
3.    for( int j = 0 ; j < yImg ; j ++){
4.      meanShift( i, j, imageMatrix[ i * yImg + j]);
5.      if(i == 0 && j == 0)numCores = omp_get_num_threads();
6.  }
```

Výpis 2 Ukázka pararelizace CPU kódu

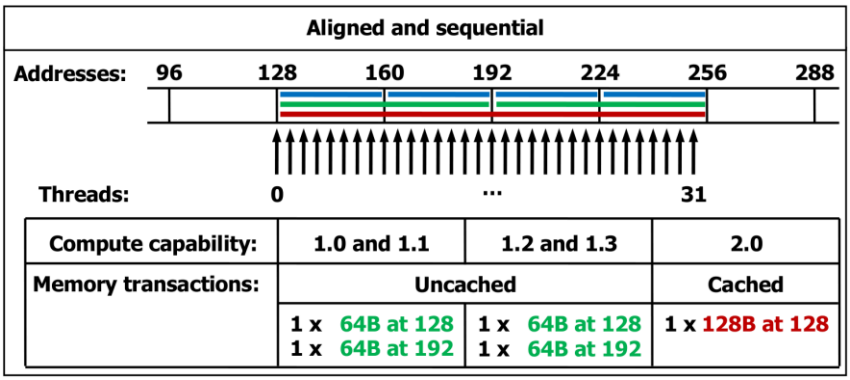
5.5 CUDA implementace

5.5.1 Rozdíl implementace CPU x GPU

Implementace programu v prostředí NVIDIA CUDA se od běžného programu odlišuje v tom, že probíhá paralelně na stovkách vláken. Problém však spočívá v tom, že tato vlákna mezi sebou nemohou komunikovat. Jediná možnost komunikace je pouze skrz paměť, kterých je však více druhů, viz kapitola 4.5. Mějme modelovou situaci, kde máme pole čísel a chceme zjistit největší prvek tohoto pole. V prostředí jazyka C toho dosáhneme tak, že si nejprve uložíme první číslo tohoto pole, které následně procházíme po jednom prvku v cyklu a porovnáváme velikost tohoto prvku s ostatními. Při nalezení většího čísla je uložíme, až nám zůstane uložen největší prvek. Programový model CUDA pracuje odlišným způsobem. Cyklus se rozbije tak, že se ke každému prvku přistupuje v jednom vlákně. Při volání kernelu se definuje počet vláken, tedy velikost pole. Nyní však nastává problém, jelikož máme n prvků v n vláknech. Tady přichází na řadu blok a sdílená paměť. V rámci bloku vlákna sdílejí rychlou sdílenou paměť a je možné je synchronizovat. Takže ve výsledku bude algoritmus nalezení největšího čísla vypadat tak, že se nalezne největší prvek v jednom bloku vláken pomocí synchronizace a rychlé sdílené paměti. Pak už jen porovnáváme výsledky bloků s hodnotou uloženou v globální paměti. Z tohoto popisu je patrné, že se tyto dva přístupy značně liší a při programování na platformě CUDA je důležité mít hlubší znalosti o fungování hardwaru.

5.5.2 Kernel 1

V tomto případě jsem se rozhodl implementovat Mean Shift algoritmus tak, že každé jednotlivé vlákno bude vypočítávat iterace pro každý jednotlivý pixel obrazu. V tomto případě je algoritmus velice podobný tomu popsanému v kapitole 5.4.2. Všechna vlákna fungují nezávisle na sobě, a proto odpadají problémy se synchronizací. Úzké hrdlo tohoto programu je v tom, že každé vlákno si načítá samostatně pixely okolí zpracovávaného bodu z globální paměti a mnohdy vlákna čtou ze stejných míst. Ideální přístup vypadá tak, že se paralelně přistupuje k jednotlivým prvkům, které se pak zpracovávají, jak je patrné z obrázku č. 5. Problém však spočívá v tom, že v každé iteraci se střed výpočetního okna posouvá. Vlákna potom cestují v obraze za shluky, a není proto možné paralelně načítat z globální paměti. Odlišnost oproti implementaci na CPU je v tom, že jsem se pokusil minimalizovat počet podmínek.



Obrázek 8 Optimální načítání z paměti

Po několika zkouškách se na mém GPU (geforce 650ti) ukázala jako nejrychlejší velikost bloku 16x16 vláken. Dvourozměrný kernel jsem zvolil proto, že je pro výpočet nových souřadnic potřeba znát x a y souřadnici zpracovávaného bodu. Vzhledem k tomu, že je malá pravděpodobnost, že bude zpracovávaný obraz mít x a y rozměr dělitelný šestnácti, je tato situace ošetřena.

```
1.  dim3 threadsPerBlock(16, 16);
2.  int xBlock = (hodnoty->x / threadsPerBlock.x);
3.  if( hodnoty->x % threadsPerBlock.x ) xBlock++;
4.  int yBlock = (hodnoty->y / threadsPerBlock.y);
5.  if( hodnoty->y % threadsPerBlock.y ) yBlock++;
6.  dim3 numBlocks( xBlock , yBlock );
7.
8.  doMeanShift<<<numBlocks, threadsPerBlock>>>(img_d, result_d);
```

Výpis 3 Ukázka volání kernelu

5.5.3 Kernel 2

U tohoto kernelu jsem se pokusil rozbít co nejvíce cyklů. Rozhodl jsem se postupovat tak, že v každém jednotlivém bloku vláken se bude zpracovávat právě jeden zdrojový pixel a vypočítávat pro něj iterace. Souřadnice zpracovávaného pixelu jsou dány vektorem blockIdx. Z tohoto vyplývá, že blok vláken reprezentuje výpočetní okno a z toho plynou určitá omezení. Tím největším je to, že je omezen poloměr velikosti výpočetního okna na hodnotu blockDim.x/2 -1. Při zvolení většího poloměru není výsledek segmentace korektní. Dalším problémem je, že vlákna vypočítávají dílčí sumy, které je třeba centrálně sečíst. V neposlední řadě je třeba mezi všemi vlákny bloku sdílet informace o nově vypočtených hodnotách středu výpočetního okna. Poslední úskalí spočívá v tom, že po dosáhnutí konvergence je třeba všechna vlákna bloku ukončit.

Tyto problémy jsem vyřešil pomocí proměnných uložených ve sdílené paměti. Před první iterací se první vlákno každého bloku postará o inicializaci sdílených proměnných, tedy souřadnic a jasu středu výpočetního okna. Po inicializaci se zavolá __syncthreads(), aby se zaručilo, že všechna vlákna mají k těmto hodnotám přístup. Potom každé vlákno bloku načte jednu hodnotu z globální paměti na základě své pozice v bloku – vektor threadIdx. Následně se provede výpočet dílčích sum a provede synchronizace bloku. Na základě těchto hodnot provede první vlákno výpočet posunu a v případě, že je překročen maximální počet iterací, nebo posun je menší než minimální definovaný, uloží výsledek a nastaví sdílenou proměnnou, která značí dosažení konvergence na true. Po tomto úkonu se vlákna znovu synchronizují. Po synchronizaci všechna vlákna bloku kontrolují ukončovací proměnnou. Je-li true, všechna vlákna skončí. Když se všechna vlákna pokouší číst ukončovací proměnnou, provádí se to takzvaným broadcastem a nedochází proto ke ztrátám výkonu. Nevýhodou je, že všechna vlákna bloku musí čekat na první vlákno, až provede své operace, které nelze paralelizovat, a proto jsou všechna vlákna bloku vyjma jednoho v nečinnosti.

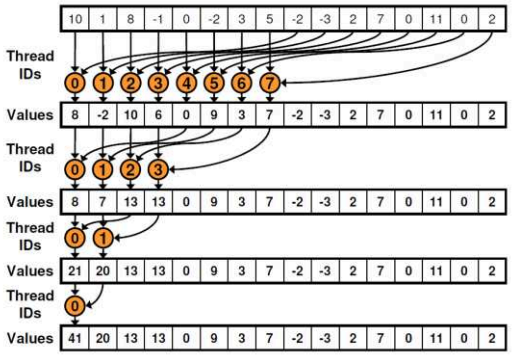
```
1.  if (blockPos == 0) {
2.
3.    xSum_sh[0] /= wSum_sh[0];
4.    ySum_sh[1] /= wSum_sh[0];
5.    zSum_sh[2] /= wSum_sh[0];
6.
7.    posun = sqrt(
8.    (xSum_sh[0] - x_sh) * (xSum_sh[0] - x_sh)
9.      + (ySum_sh[1] - y_sh) * (ySum_sh[1] - y_sh)
10.       + (zSum_sh[2] - rootColor_sh)
11.         * (zSum_sh[2] - rootColor_sh));
12.
13.   if (i >= maxIteraci || posun < minPosun) {
14.     res[xD * hodnotyD.y + yD] = zSum_sh[0];
15.     forEnd_sh = true;
16.   }
17.
18.   x_sh = xSum_sh[0];
19.   y_sh = ySum_sh[0];
20.   rootColor_sh = zSum_sh[0];
21. }
22. __syncthreads();
```

Výpis 4 Ukázka operací prvního vlákna bloku

Každé vlákno bloku vypočítá sumu pixelu, který načte. Tyto sumy je ale potřeba sečíst. K tomuto účelu jsem využil paralelní redukce. Jedná se o operaci, v mém případě sčítání, kdy z n vstupních hodnot redukuje na jednu výstupní. Operace musí být asociativní, což sčítání splňuje. Redukce funguje v několika krocích, kdy v každém dalším kroku se půlí zpracovávaný interval, až zůstane jen jeden prvek, jak je vidět na obrázku č. 6. Nevýhoda je v tom, že se už při prvním součtu využívá jen polovina vláken, a v každém dalším kroku se počet aktivních vláken snižuje na polovinu. Protože není zaručeno provedení všech vláken v bloku současně, je vhodné, aby pracovala všechna vlákna warpu. Toho jsem docílil tím, že sčítání se účastní prvních 128 vláken, pak 64 atd.

```
1.  for (int s = grid * grid / 2; s > 0; s >>= 1) {
2.    if (blockPos < s) {
3.      wSum_sh[blockPos] += wSum_sh[blockPos + s];
4.      xSum_sh[blockPos] += xSum_sh[blockPos + s];
5.      ySum_sh[blockPos] += ySum_sh[blockPos + s];
6.    zSum_sh[blockPos] += zSum_sh[blockPos + s];
7.    }
8.    __syncthreads();
9.  }
```

Výpis 5 Ukázka součtu sum



Obrázek 9 Pararelní součet

5.6 Nekonečný kernel

Pro demonstraci optimálního využití sdílené paměti jsem použil nekonečný kernel. Termínem nekonečný se myslí tak velký kernel, který pokryje všechny body v obraze. Z toho vyplývá jeho velká výpočetní náročnost. Zvolil jsem Gaussianuv kernel, protože jeho definiční obor nemá žádná omezení, a proto se budou všechny načtené body zahrnovat do výpočtu. O výsledném tvaru kernelu budou rozhodovat, stejně jako u konečného kernelu, poloměr výpočetního okna h a maximální jasová vzdálenost.

5.6.1 CPU implementace

Algoritmus je velice podobný tomu, který je popsán v kapitole 5.4.2. Hlavní rozdíl je v tom, že se v každé iteraci musí procházet celý obraz. Odpadají proto podmínky nutné k ošetření okrajů obrazu a výpočet pozice nového výpočetního okna. V každé iteraci se vypočítají nové souřadnice $[x, z, y]$. Tyto souřadnice se pak využijí v iteraci následující jako střed kernelu. K paralelizaci jsem využil, stejně jako v případě z kapitoly 5.4.2, OpenMP.

```
1.  void meanShift( int xP, int yP, uint8_t color ){
2.   //proměnné
3.   while(true){
4.     //podmínky okraje obrazu a výpočetního okna
5.     for( int i = 0 ; i < xRozmerObr ; i++ )
6.     {
7.       for( int j = 0 ; j < yRozmerObr; j ++)
8.       {
9.         //výpočet sum
10.      }
11.    }
12.    //ukončení algoritmu?
13.    if( iterace > maxIteraci || posun < minPosun ){
14.      //ulož výsledek
15.      break;
16.    }
17.    //nastaveni nových hodnot středu výpočetního okna
18.  }
19. }
```

Výpis 6 Ukázka algoritmu

5.6.2 CUDA implementace 1

V této implementaci je algoritmus velice podobný CPU implementaci. V každém vlákně se počítají iterace pro jeden zdrojový bod obrazu, dokud nedosáhne konvergence. Každé vlákno si načítá hodnoty jasu jednotlivých bodů z globální paměti. Při použití nekonečného kernelu je však dopředu známá pozice načítaných bodů, protože se vždy načítá celý obraz. Proto se jedná o neefektivní načítání z globální paměti, jelikož všechna vlákna načítají stejné hodnoty. Dává nám to tedy prostor pro optimalizace.

```
1.  //...
2.  for (int i = 0; i < hodnotyD.x; i++) {
3.    for (int j = 0; j < hodnotyD.y; j++) {
4.      childColor = img[i * hodnotyD.y + j];
5.      //výpočet sum
6.    }
7.  }
8.  //...
```

Výpis 7 Ukázka načítání z globální paměti

5.6.3 CUDA implementace 2

V této implementaci jsem zohlednil to, že vlákna v rámci jednoho bloku mezi sebou sdílejí rychlou sdílenou paměť. Algoritmus funguje podobně jako v předchozí kapitole. V každém vlákně se počítají iterace s nekonečným výpočetním oknem. Každé vlákno proto ve všech iteracích načítá celý obraz, a vypočte nový střed kernelu. V další iteraci pracuje s tímto středem, dokud nedosáhne konvergence.

Protože všechna vlákna mají stejné výpočetní okno, a tím je celý obraz, dopředu víme, které hodnoty budou potřeba k výpočtu. V tomto momentě jsem využil sdílenou paměť. V rámci jednoho bloku, každé vlákno načte jednu hodnotu z globální paměti. Kterou hodnotu vlákno načte je určeno vektorem threadIdx. Načtená hodnota se uloží do sdílené paměti a pak se vlákna synchronizují. Vzápětí všechna vlákna v bloku procházejí tuto sdílenou paměť bez nutnosti přístupu do globální paměti. Tímto dojde ke zrychlení algoritmu.

```
1.  //...
2.  int xT = threadIdx.x;
3.  int yT = threadIdx.y;
4.  //...
5.    for (int i = 0; i < hodnotyD.x; i += grid) {
6.      for (int j = 0; j < hodnotyD.y; j += grid) {
7.        buffer_sh[xT][yT] = img[(i + xT) * hodnotyD.y + (j + yT)];
8.        __syncthreads();
9.        for (int a = 0; a < grid; a++) {
10.         for (int b = 0; b < grid; b++) {
11.           childColor = buffer_sh[a][b];
12.           //výpočet sum
13.       }
14.       //...
15.     }
16.   }
17. //...
```

Výpis 8 Ukázka načítání do sdílené paměti

5.7 Problémy

Problémem při výpočtu takto náročného algoritmu na GPU je, že pokud je grafická karta použita k zobrazování grafického výstupu na monitor, OS výpočet po překročení určitého času ukončí. Proto, chceme-li tento problém obejít, je třeba si překonfigurovat OS tak, aby algoritmus doběhl dokonce. Toho se týká jedna nepříjemnost při ladění programu na GPU, protože v průběhu výpočtu PC nereaguje na uživatelské vstupy. Proto v momentě, kdy je v programu chybná nekonečná smyčka, je nutné PC restartovat. Dalším rozdílem oproti klasickému prostředí jazyka C/C++ je ladění. Není úplně jednoduché odhalit chybu v aplikaci, která běží ve stovkách vláken. Ke zjednodušení debuggingu CUDA v nejnovější verzi podporuje funkci printf(), která mi při ladění mých programů hodně pomohla.

6 Dosažené výsledky

6.1 Testovací podmínky

Všechna měření jsem prováděl na sestavě uvedené v tabulce. Programy jsem spouštěl s nejvyšší prioritou za pomocí unixového programu nice. Rozhodl jsem se tak proto, aby byly výsledky co nejméně ovlivněny systémovými procesy. Všechna měření s využitím pouze CPU jsem prováděl na všech čtyřech jádrech procesoru, jelikož v dnešní době se už vyskytují i v mobilních zařízeních dvou a více jádrové procesory, a proto je celkový výkon CPU zajímavější než jen jeho části.

Tabulka 1 Testovací sestava

Procesor	AMD Athlon II X4 620, 4x2.6 GHz
Grafická Karta	NVIDIA GeForce GTX 650 Ti
RAM	12GB DDR3 1333 MHz
OS	Ubuntu 12.04 LTS 64 bit
Verze gcc	gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
Verze nvcc	release 5.0, V0.2.1221

6.2 Výsledky segmentace

Jak jsem uvedl v kapitole 3, Mean Shift segmentace má tři argumenty, které zásadně ovlivňují výkon i kvalitu segmentace. Na obrázku č. 10 vidíme výsledky segmentace s použitím nekonečného Gaussianova kernelu. Hodnoty pod jednotlivými obrázky reprezentují poloměr výpočetního okna a maximální jasovou vzdálenost.

Obrázek 10 Segmentace s různými parametry

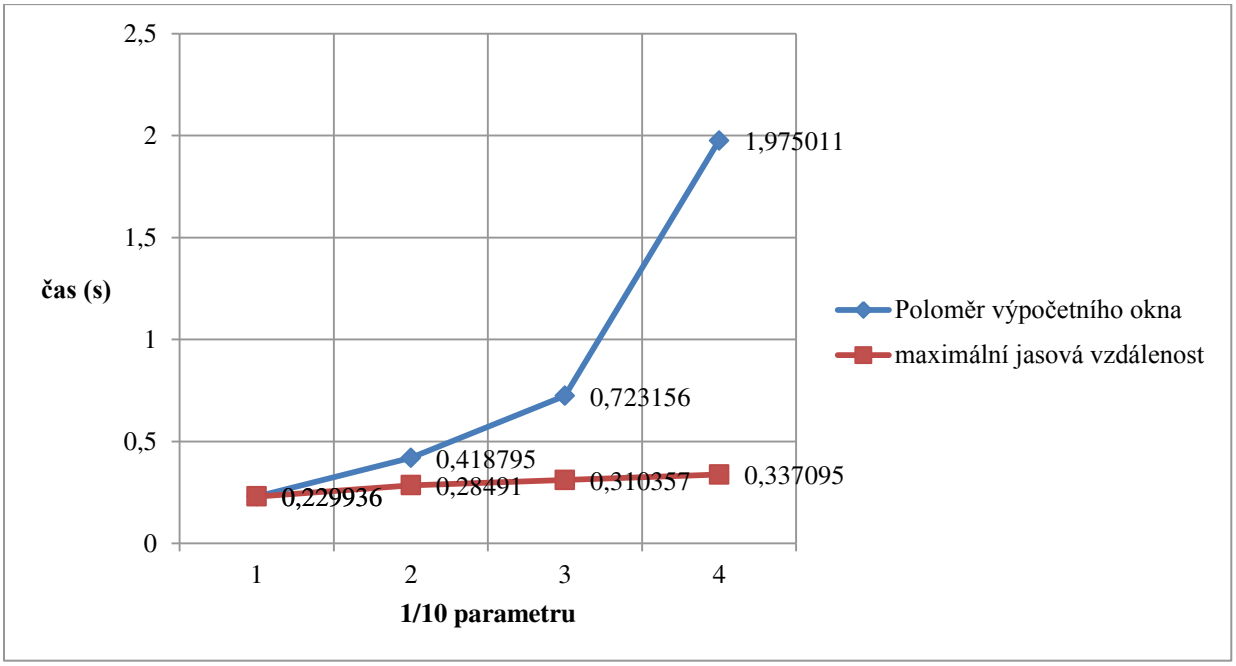
Jak si můžeme všimnout, s rostoucí velikostí *poloměru výpočetního okna* ubývá detailů, ale tvar segmentů zůstává stejný. To můžeme vidět například jako rozdíl mezi obrázky s hodnotami 5,10 a 20,10. U stromu 5,10 vidíme tmavá místa, na obrázku 20,10 se nevyskytují. Na druhou stranu s rostoucí velikostí *jasové vzdálenosti* můžou splývat sousedící segmenty. Toto vidíme u obrázků s parametry 20,40, kde se na rozdíl od jiných obrázků vytratila levá část stromu.

Z tohoto je patrné, že volba správných argumentů má zásadní vliv na výsledek segmentace.

6.3 Vliv parametrů na výkon

Jak jsem uvedl v kapitole 3.4, Mean Shift segmentace má parametry, které významně ovlivňuji výkon segmentace. Výsledky kvality s různými parametry jsou uvedeny v předchozí kapitole. Vliv na výkon segmentace vidíme na obrázku 13. K měření jsem použil program popsaný v 5.5.2, tedy konečný Epanechnikův kernel. Konvergence byla nastavena na minimální posun 0.001 pixelu, nebo dosažní 500 iterací. Měření jsem prováděl na obrázku velikosti 128x128 pixelů.

Vodorovná osa reprezentuje čas v sekundách, svislá osa pak hodnotu desetiny parametru. U křivky *poloměr výpočetního okna* hodnota na svislé ose reprezentuje poloměr výpočetního okna, u *maximální jasové vzdálenosti* pak jasovou vzdálenost.

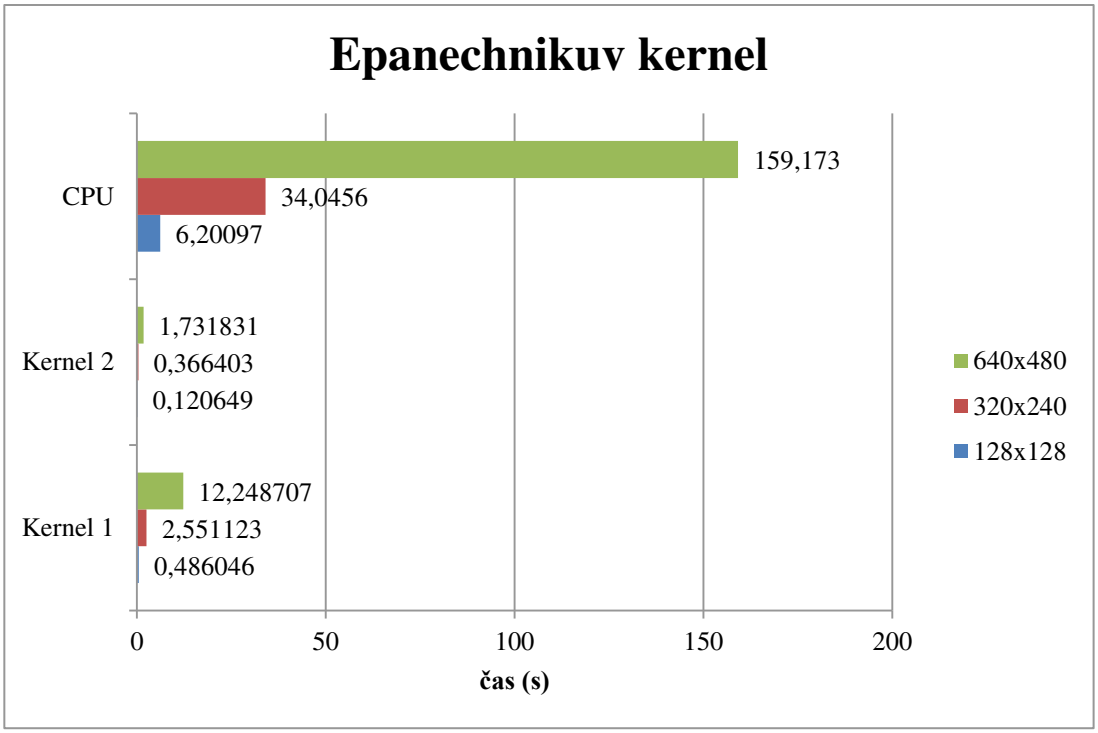


Obrázek 11 Vliv parametrů na výkon

Jak je patrné, tak s rostoucím poloměrem výpočetního okna roste doba průběhu algoritmu exponenciálně. S rostoucí jasovou vzdáleností roste doba vykonání algoritmu lineárně.

6.4 Výkon konečného Epanechnikova kernelu

Na obrázku č. 12 vidíme výsledky Mean Shift segmentace s použitím konečného Epanechnikova kernelu. Vstupními parametry pro měření jsem zvolil: poloměr výpočetního okna 7 a maximální jasová vzdálenost 20. Poloměr 7 jsem zvolil proto, že jsem použil velikost bloku vláken 16x16. Z toho vyplývá, že u kernelu 1 by při volbě většího poloměru výpočetního okna nebyl výsledek segmentace korektní. Konvergenci jsem nastavil po dosažení 500 iteraci, nebo je-li posun menší než 0.001bodu pixelu.



Obrázek 12 Výkon Epanechnikova kernelu

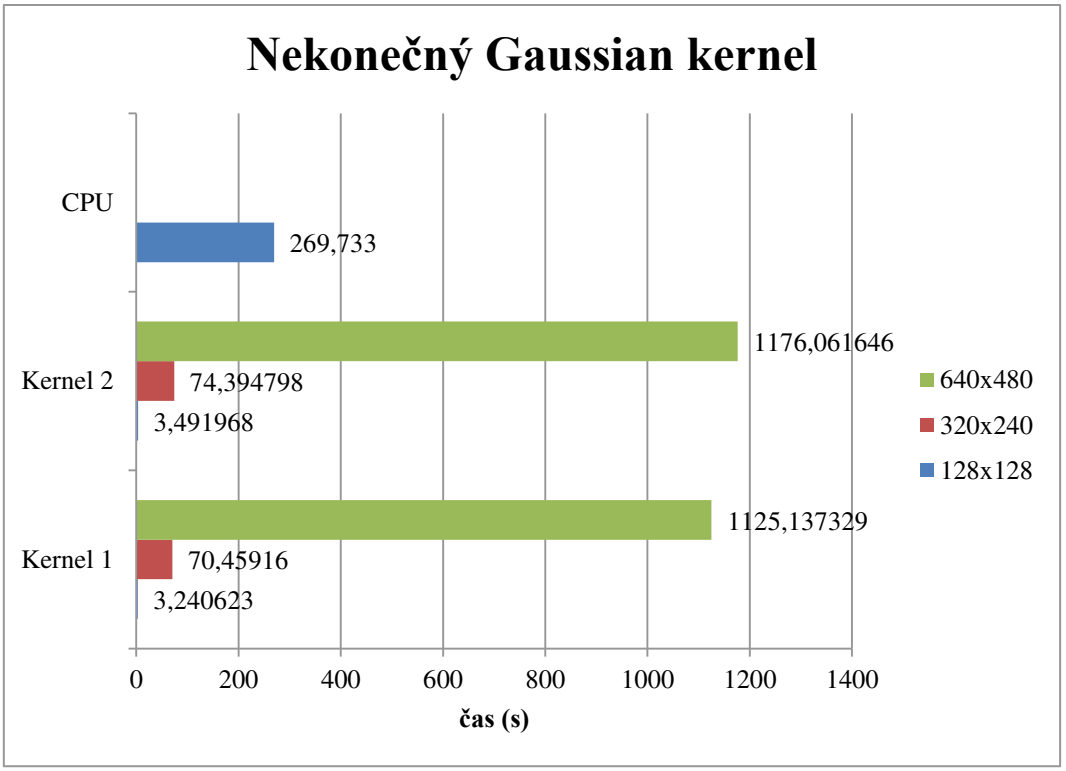
Kernel 1 na obrázku 12 reprezentuje program z kapitoly 5.5.3, u kterého je využitá sdílená paměť, kernel 2 pak program z kapitoly 5.5.2, který načítá data pouze z globální paměti.

Vidíme velký výkonový rozdíl mezi kernelem 1a 2. Ten je dán tím, že zatímco u kernelu 2 je úzké hrdlo programu pouze načítání hodnot z globální paměti, u kernelu 1 je problém se sdílenými proměnnými. Problém je v tom, že většina vláken je při výpočtu ve stavu čekání na synchronizaci ostatních vláken. Je to jednak tím, že při paralelním sčítáni, se už při prvním kroku využije jen polovina vláken a v dalších krocích se počet využitých vláken snižuje na polovinu. Dalším problémem je, že všechna vlákna bloku musí čekat na jedno vlákno, než provede výpočet posunu a aktualizuje sdílené proměnné, což je další degradace výkonu. Z tohoto vyplývá, že implementace kernelu 1 je neefektivní, poněvadž se značná část výpočtu čeká. Z grafu je ale patrné, že i neefektivně napsaný program spuštěný na GPU je několikanásobně rychlejší než ekvivalentní program na CPU.

V grafu vidíme, že dobře napsaný algoritmus pro GPU je více než 90x rychlejší než na čtyřjádrovém CPU. Toto vidíme, jako rozdíl mezi CPU a Kernelem 2, u obrázků 320x240 a 640x480. Můžeme si také všimnout, že i neefektivně napsaný program v NVIDIA CUDA je o řád rychlejší než ekvivalentní program pro CPU.

6.5 Výkon nekonečného Gaussianova kernelu

Na obrázku č. 9 vidíme výsledky Mean Shift segmentace s použitím nekonečného Gaussianova kernelu. Pro měření jsem zvolil poloměr výpočetního okna 10 a maximální jasovou vzdálenost 20. Použití nekonečného kernelu je velice výpočetně náročné, a proto jsem provedl měření pro CPU jen pro obrázek 128x128 pixelů. Výpočet na CPU pro obrázek velikosti 320x240 by trval až několik hodin, a proto jsem se rozhodl jej neměřit.



Obrázek 13 Výkon Nekonečného Gaussianova kernelu

Kernel 1 je s použitím sdílené paměti popsaný v kapitole 5.6.2. Kernel 2 je s využitím pouze globální paměti popsaný v kapitole 5.6.3.

U tohoto měření jsem určil, že konvergence algoritmus dosáhne po padesáti iteracích. Jeden z důvodu této volby byla časová náročnost algoritmu. Hlavním důvodem je však problém u kernelu 1. Tento kernel využívá sdílené paměti tak, že všechna vlákna v rámci bloku načtou data do sdílené paměti, se kterými pak pracují. Odpadá tak opakované načítání dat z globální paměti. Problém ale nastává v momentě, když jsou ukončeny iterace při překročení hranice minimálního posunu. V tomto případě musí vlákna po dosažení konvergence stále načítat data do sdílené paměti a čekat na poslední vlákno v bloku než dosáhne konvergence. Vlákna, která dosáhla konvergence, pak jen načítají data a provádějí prázdné výpočty, což zpomaluje algoritmus. Při nastavení konvergence na konstantní počet iterací tento problém nenastává, protože všechna vlákna v rámci bloku dosáhnou konvergence ve stejné době. V kernelu 2 tento problém s konvergencí nepozorujeme, jelikož každé vlákno si načítá svá data, a je proto nezávislé na ostatních. Pro nastavení konstantního počtu iterací jsem se rozhodl, abych ukázal zrychlení algoritmu při využití sdílené paměti.

Z výsledků vidíme, že se algoritmus při využití sdílené paměti zrychlil o 5 - 7 %. Zrychlení oproti CPU je cca osmdesátinásobné.

7 Závěr

Cílem této bakalářské práce bylo nastudování segmentace obrazu metodou Mean Shift a vytvoření programu pro výpočet této metody. Praktická část této práce byla věnována implementaci tohoto algoritmu v prostředí jazyka C/C++ a využití masivního paralelismu grafických karet pomocí technologie NVIDIA CUDA.

První část této práce se zabývá obecným popisem segmentace obrazu. Dále je pak uveden stručný popis několika vybraných metod segmentace. Další část práce je věnována popisu Mean Shift segmentace obrazu. Je zde popsán detailní průběh segmentace a shrnutí potřebných znalostí nutných k implementaci tohoto algoritmu. Následně se práce zabývá implementacemi algoritmů v různých prostředích a srovnáváním výsledků těchto implementací.

Mean Shift algoritmus se při implementaci v prostředí klasického jazyka C/C++ ukázal velice pomalý. Přestože jsem využil technologie OpenMP a čtyř jádrového procesoru, kdy byla v průběhu výpočtu všechna jádra procesoru vytížena na 100%, běžel algoritmus velice dlouho. Proto se ukázalo využití masivního paralelismu pomocí technologie NVIDIA CUDA jako velice vhodné.

Při implementaci v prostředí NVIDIA CUDA jsem dosáhl přibližně 80 – 90násobné zrychlení algoritmu oproti čtyřjádrovému CPU. Po upravení parametrů algoritmu tak, aby se dala využít sdílená paměť GPU, jsem dosáhl dalšího 5 – 7% zrychlení oproti již tak rychlé CUDA implementaci. Z toho vyplývá vysoká efektivita výpočtů v prostředí NVIDIA CUDA.

V této práci jsem se poprvé seznámil s technologiemi OpenMP a NVIDIA CUDA. Díky těmto technologiím jsem si osvojil programování masivně paralelních aplikací, které mi v budoucnu umožní psát efektivnější programy, třeba i s využitím GPU. Mým dalším poznatkem je složitější debugging paralelních aplikací a s tím spojená větší časová náročnost odladění paralelního algoritmu. S tím souvisí i nutnost znát podrobně cílovou platformu pro napsání co nejefektivnějšího algoritmu.

8 Literatura

[1] NVIDIA C Programming guide

[2] http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[3] http://courses.csail.mit.edu/6.869/handouts/PAMIMeanshift.pdf

[4] http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

[5] http://en.wikipedia.org/wiki/Image_segmentation